TÉCNICO LISBOA

# Deduplication vs Privacy Tradeoffs in Cloud Storage

**Rodrigo de Magalhães Marques dos Santos Silva**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisors: Prof. Miguel Nuno Dias Alves Pupo Correia
Prof. Luís Eduardo Teixeira Rodrigues

### Examination Committee

Chairperson: Prof. José Alberto Rodrigues Pereira Sardinha
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Marcelo Pasin

**November 2022**

# Acknowledgments

First, I would like to acknowledge my dissertation supervisors Prof. Luís Rodrigues and Prof. Miguel Correia for their insight, support, and sharing of knowledge that has made this Thesis possible. I would also like to thank Cláudio Correia and Rafael Soares for the fruitful discussions that we had during the course of the thesis.

Also, a huge thank you to all my friends and colleagues who helped me grow as a person and who were always there for me during the good and bad times of my life. I would leave a special thanks to Rui Paças, Beatriz Feliciano, and Francisco Silva for always having my back, and supporting me when times were rough.

I would also like to thank my girlfriend Catarina Cavique and her family for their help and support during the making of this thesis.

Last but not least, I would like to thank my family, especially my parents and my sister for their friendship, encouragement, and care over all these years, for always being there for me through thick and thin, and without whom this project would not be possible.

To each and every one of you – Thank you.

# Declaration

*I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Pratices of the Universidade of Lisboa.*

# Abstract

To ensure the privacy of their data when stored in the cloud, users can choose to encrypt files before exporting them. Unfortunately, without additional mechanisms, encrypted data storage makes it impossible to implement server-side deduplication techniques, as two identical files will have different encrypted versions. In this thesis, we address the problem of reconciling the need to encrypt data with the advantages of deduplication. In particular, we study techniques that achieve this objective while avoiding frequency analysis attacks (an attack that allows an adversary to infer the content of an encrypted file based on how frequently the file is stored and/or accessed). We propose a new protocol to assign encryption keys to files that leverages the use of trusted execution environments to hide the frequency of a file from the adversary.

# Keywords

Privacy; Deduplication; Cloud Computing.

# Resumo

Para garantir a privacidade dos seus dados quando estes são armazenados na nuvem, os utilizadores podem optar por cifrar os ficheiros antes de os exportar. Infelizmente, sem mecanismos adicionais, o armazenamento de dados cifrados inviabiliza a concretização de técnicas de deduplicação do lado do servidor, pois dois ficheiros idênticos terão versões cifradas distintas. Nesta tese abordamos o problema de conciliar a necessidade de cifrar os dados com as vantagens da deduplicação. Em particular, estudamos técnicas que atingem este objetivo, evitando ataques de frequência (um ataque que permite inferir qual o conteúdo cifrado com base na frequência com que este é acedido). Neste contexto, propomos um novo protocolo para escolher as chaves de cifra, que tira proveito do uso de ambientes de execução seguros para esconder a frequência de um ficheiro de um atacante.

# Palavras Chave

Privacidade; Deduplicação; Computação na Nuvem.

# Contents

# List of Figures

# List of Tables

# Acronyms

| | |
|---|---|
| **CE** | Convergent Encryption |
| **CM-SKETCH** | Count-Min Sketch |
| **DHKE** | Diffie-Hellman Key Exchange |
| **DUPLESS** | Duplicateless Encryption for Simple Storage |
| **EPC** | Enclave Page Cache |
| **HCE1** | Hash and Convergent Encryption 1 |
| **HCE2** | Hash and Convergent Encryption 2 |
| **ISCSI** | Internet Small Computer System Interface |
| **KS** | Key Server |
| **MLE** | Message-Locked Encryption |
| **OPRF** | Oblivious Pseudorandom Function |
| **POW** | Proof-of-ownership |
| **RCE** | Randomized Convergent Encryption |
| **S2DEDUP** | SGX Enabled Secure Deduplication |
| **SGX** | Software Guard Extensions |
| **SKE** | Symmetric-Key Encryption |
| **SPDK** | Software Performance Development Kit |
| **SS** | Storage Service |
| **STC** | Strong Tag Consistency |
| **TC** | Tag Consistency |
| **TED** | Tunable Encrypted Deduplication |
| **TEE** | Trusted Execution Environment |

# 1

# Introduction

## Contents

## 1.1 Motivation

When users backup their computers, different users are likely to backup the same files, such as software binaries and music files, among others. It has been observed that, in systems that store large amounts of data from multiple users, such as Dropbox [1] and Google Drive [2], it is possible to find large amounts of repeated data. For instance, when users store program sources in the cloud, there are many libraries that may be shared by several projects. *Data Deduplication* is a technique that allows one to identify files that are identical or that have identical pieces (also known as *chunks*), and treat all copies of the same chunk as a single unit. In this way, the system needs to maintain just a few copies of each unique chunk, enough to provide fault tolerance, regardless of the number of users that store that chunk.

Unfortunately, it is hard to apply data deduplication to encrypted data. If the users encrypt their data before storing it and use different keys to perform encryption, the same chunk will result in different encrypted versions. This prevents the storage system from identifying which data is redundant and, therefore, from applying data deduplication. *Encrypted Deduplication* is the name given to techniques that attempt to combine file encryption and data deduplication. Typically, this combination requires some form of direct or indirect coordination among different users to ensure that identical chunks are encrypted with the same keys. The challenge is to perform this coordination in a manner that is both efficient and allows to preserve the confidentiality[1] of the information stored by each user.

## 1.2 Contributions

In this thesis, we survey the main techniques that have been proposed to conciliate deduplication and the possibility of storing data in an encrypted form, in a manner that can ensure data confidentiality for the clients. Inspired by these works, we propose an architecture that leverages Trusted Execution Environments (TEEs) to increase the performance of encrypted deduplication systems. In detail:

- The thesis describes a novel tunable encrypted deduplication system that leverages trusted execution environments to perform sensitive cryptographic operations while keeping track of the frequency of all chunks by using an internal frequency table (cache), and a larger external encrypted table in an untrusted environment.

Our mechanism that accesses the table in the untrusted environment allows the use of multiple access policies, and cache eviction policies. Each one provides different performance and security tradeoffs.

[1] We use the term *confidentiality* with the same meaning as [3]. i.e, "*... to protect outsourced storage from the unauthorized access by malicious users or even the cloud providers that host the outsourcing services*".

We are the first to offer full privacy protection while achieving exact deduplication, we name our system FH-Dedup since we provide secure deduplication through Frequency Hiding. We propose different techniques to protect our vulnerable external table, stored outside the TEE. Additionally, our proposed protocol allows clients to read their files without requiring interaction with the TEE, offering an increased performance on reading operations, more than 5x faster.

## 1.3 Results

This thesis has produced the following results:

- An implementation of FH-Dedup, a tunable encrypted deduplication system with our access mechanism to an external encrypted frequency table for devices that support *Intel Software Guard Extensions (SGX)*;
- An extensive experimental evaluation of FH-Dedup, covering the performance of read and write operations, the deduplication gain, and the privacy properties offered by our solution.

## 1.4 Research History

Parts of the work described in this thesis have been published as:

- R. Silva, C. Correia, M. Correia and L. Rodrigues. Ataques de Frequência em Deduplicação Cifrada na Nuvem. In *Actas do décimo terceiro Simpósio de Informática (Inforum)*, Guarda, Portugal, September 2022.

This work was supported by FCT – Fundação para a Ciência e a Tecnologia, through the grant 2020.05270.BD, the NGSTORAGE project (funded by the OE with the ref. PTDC/CCI-INF/32038/2017), and the project UIDB/50021/2020.

## 1.5 Organization of the Document

The rest of the thesis is organized as follows: Chapter 2 introduces the main concepts relevant to our work and Chapter 3 describes the main techniques to implement encrypted deduplication. Chapter 4 describes the system architecture and implementation. Chapter 5 shows our evaluation of the system. Finally, Chapter 6 concludes the thesis and highlights directions for future work.

# 2

# Background

## Contents

This chapter introduces concepts relevant to our work. We start by introducing deduplication (Section 2.1) and encrypted deduplication (Section 2.2), then we list the main attacks against encrypted deduplication (Section 2.3), then we present secure mediation (Section 2.4), and finally, we introduce TEEs, and their security mechanisms and properties (Section 2.5).

## 2.1 Data Deduplication

Data deduplication is a high-coarse grained compression technique that prevents the same content from being stored with a much larger redundancy degree than strictly needed since systems only need to store enough copies to provide fault tolerance and create pointers to the content. Deduplication identifies duplicated data at the granularity of files or a portion of files, denoted as file *chunks*. If two or more files, from the same or different users, share a given chunk, the system needs to maintain just a few copies of that chunk, enough to provide fault tolerance, regardless of the number of files that include it.

Unlike traditional compression techniques that eliminate redundancies within a file or a small group of files (usually stored in the same operation), data deduplication aims to eliminate large data sets stored at different times by uncoordinated users and activities [4].

The effectiveness of deduplication is measured by the deduplication gain. The deduplication gain is defined as the number of duplicates that are actually eliminated, thereby reducing the storage space required to store some piece of data. Studies have shown that data deduplication can achieve significant storage savings in a production environment, for example, saving 50% on primary storage [5] and up to 98% on backup storage [6].

Another way to save storage space using deduplication is the use of *cross-user* deduplication. This means that each file or chunk is compared with the data of other users, and if the same copy is already on the server, data deduplication is performed.

This method is popular because it saves storage and bandwidth, not only when a single user has multiple copies of the same data but also when different users store copies of the same data.

### 2.1.1 Chunks

The unit of data deduplication is called a chunk. The size of a chunk and the technique used to divide data into chunks affect the performance of the system [4].

One of the most straightforward ways to divide data into chunks is to consider file boundaries [7], which is known as whole file chunking. Another simple approach consists of dividing files into fixed-sized chunks (e.g., 4-8 KB each). Unfortunately, none of these approaches is very effective at producing chunks that can be deduplicated. For instance, consider two files, where one file only differs from the

other by a single, additional, byte placed at the beginning of the file. Despite that most of their content is the same, the two files are different and their fixed-sized chunks would also be all different. This is known as the boundary-shifting problem [8].

Several algorithms to divide files into chunks have been proposed in the literature [9–11]. These techniques are orthogonal to our work.

### 2.1.2 Deduplication Approaches

To achieve deduplication, one should match existing stored data with the data the client wants to store, if there is a match, deduplication can be performed. This can either happen at the client-side or at the server-side.

In *source-based deduplication* the client is responsible for checking if the data can be deduplicated before sending it to the server. For this purpose, the client queries the Storage Service (SS) about existing duplicated data and only uploads the missing data to the SS. A challenge in this technique is that the client needs to prove that it owns the data before being given access to the deduplicated copy. In addition, this approach leaks information to clients about the existence of other clients with the same data.

In *target-based deduplication* the client always sends the data to the SS, which is responsible for performing deduplication. This prevents the client from extracting information about which files are deduplicated, and also allows the SS to trivially verify that the client owns the data. However, it does not prevent deduplicated data from consuming network resources.

## 2.2 Encrypted Deduplication

Encrypted deduplication augments deduplication with support for data encryption. Typically, not only is the data stored in encrypted form by the SS, but the SS also has no access to the plaintext and cannot infer the content being stored. Support for encrypted deduplication is relevant because often the data is confidential and the clients want to keep information regarding data ownership private.

The challenge of implementing encrypted deduplication is to design schemes that will result in having the same chunks encrypted with the same keys without violating confidentiality. Notice that, if each client encrypts the chunks independently, without any sort of coordination with other clients, the same chunk from different clients would result in different encrypted chunks, preventing cross-user deduplication.

## 2.3 Privacy Attacks

Deduplication is more effective when applied across multiple users (cross-user deduplication). However, this approach has serious privacy implications, which open the doors for many attacks with the intent of getting insight about the system and the content stored by users.

### 2.3.1 Brute Force

If each user encrypts its own chunks with a different key, only known to that user, cross-user deduplication becomes almost impossible. Thus, as we will see later in the text, approaches that support cross-user deduplication have to make sure that the same files are encrypted with the same keys, even when stored by different users. If not done carefully, this may open the door for attacks, including brute force attacks.

One of the simplest strategies to ensure that different clients encrypt the same file with the same key consists in deriving the encryption key deterministically from the content of the file [12]. An example of this approach is Message-Locked Encryption (MLE) [13], which we will present in detail later in the text. This technique is vulnerable to brute force attacks, as follows:

Consider that the SS wants to check if a given user stores copies of a given book. Given that the encryption key is derived from the content, the SS can obtain a copy of the book, derive the key, encrypt the book, and compare the encrypted version with the files stored by the user. This attack can be executed even if there are several editions or versions of the book, by using this strategy for all versions.

As we shall discuss, several systems to support encrypted deduplication avoid the use of MLE, and use other techniques that are less vulnerable to brute-force attacks.

### 2.3.2 Deduplication Detection

One privacy issue that arises when dealing with systems that provide deduplication is the identification of deduplication, that is if deduplication occurred on the data that was sent to the SS. Although it may seem harmless at first, a malicious client can discover whether a given file or chunk of data was already stored. This may release personal information, so it may be a privacy problem.

Most vendors will not try to hide the fact that they use data deduplication. This can be checked by reading the upload status message, checking the upload speed to see if the file upload is completed in a much shorter time than the client computer upload bandwidth, or monitoring network traffic and measuring the amount of data transferred (this is the most common deduplication detection method). Cross-user source-based deduplication [14] is particularly vulnerable to this kind of attack.

Suppose that the attacker wants to know information about a given user. If the attacker suspects that this user owns a certain sensitive file that is unlikely to be owned by any other user, the attacker can

use deduplication to check whether this conjecture is true. All the attacker needs to do is send file *F* to the SS and check if deduplication has occurred. Dropbox [1] was vulnerable to this attack [15] [14], a malicious client could infer if other users had previously stored the same file by observing the network traffic and seeing the size of the data being transferred.

### 2.3.3 Frequency Analysis

To get insight about the data that has been stored, attackers developed techniques that allow mapping a plaintext to its ciphertext based on its frequency ranking. Frequency analysis is an inference attack that has been used to predict and recover plaintexts from substitution-based ciphertexts. A simple example of a substitution-based ciphertext is the Caesar Cipher [16], which replaces each letter in a given message by a letter in some fixed position down the alphabet. This type of attack has been shown to be useful for breaking deterministic encryption.

During a frequency analysis attack, an attacker has access to a set of plaintexts and a set of ciphertexts, and the goal is to make a relation between each ciphertext and plaintext in both sets. To launch the attack, an attacker ranks the plaintexts and ciphertexts which he has access by their frequency and then associates each ciphertext with the plaintext in the same frequency rank. In many cases, the attacker can then infer that the most frequent plaintext chunk maps to the most frequent ciphertext chunk.

After a mapping between plaintext and ciphertext is made, there are attacks that enhance the severity of classical frequency analysis attacks by exploiting the locality of chunks [17], designated *Locality-Based* attacks. Chunk locality states that chunks are likely to reoccur together with their neighbor chunks across storage backups.

A locality-based attack can occur if a plaintext chunk *M* of a prior backup was identified as the original plaintext chunk of a ciphertext chunk *C* of the latest backup, then both (left and right) neighbors of *M* are also likely to be the original plaintext chunks of the left and right neighbors of *C*, this can happen since chunk locality implies that the ordering of chunks is likely to be preserved across backups.

## 2.4 Secure Mediation

A way to achieve encrypted deduplication and prevent frequency analysis consists of resorting to a trusted entity, the *secure mediator*, that coordinates the allocation of content-encryption keys. Whenever clients need to encrypt a file, clients contact a secure intermediary, who is responsible for indicating which key must be used to encrypt each chunk. Given multiple copies of the same content, the choice of keys by the mediator allows separating these copies into different sets, within each set, the copies are encrypted with a given key (enabling deduplication), and each set uses a different key (avoiding

frequency analysis). Trust in the mediator can be achieved using cryptographic techniques and/or secure hardware.

## 2.5 Trusted Execution Environments

A TEE is a secure mode of the CPU that allows one to run code and store data isolated from the operating system and user-level processes. As the need for digital trust grows and concerns about protecting connected devices increases, TEEs become more and more important [18, 19]. The motivation to use TEEs in encrypted deduplication systems is to improve performance while still maintaining security, bandwidth efficiency, and storage efficiency, by running sensitive operations in TEE.

*Enclaves* are a type of TEE technology that has been made available in many common Intel CPUs. The *enclaves*, provided by the Intel SGX architecture [20], are TEEs that leverage hardware mechanisms such as *hardware secrets*, *remote attestation*, *sealed storage* and *memory encryption*. The hardware secrets are the root provisioning and root seal keys. Remote attestation is enforced for the client in order to prove to the service provider that an enclave is running a given software, inside a given CPU, with a given security level. Sealed storage is used to save secret data on untrusted media, required to persist data between reboots or failures because the enclave state is stored in volatile memory [20].

The enclave relies on a hardware-guarded memory region called the Enclave Page Cache (EPC) for hosting any protected code and data. An EPC comprises 4 KB pages, and any in-enclave application can use up to 128 MB. If an enclave application has a larger size than the EPC, it encrypts unused pages and evicts them to unprotected memory, suffering a performance penalty [21]. SGX provides two interfaces: ECALLs, used by an application to invoke enclave functionality, and OCALLs, used by the enclave code to access an outside application.

## Summary

In this chapter, we introduced the relevant concepts needed to understand our work: Data deduplication, encrypted deduplication, attacks against deduplication systems, secure mediation, and trusted execution environments.

In the next chapter, we present the related work in the areas of encrypted deduplication, key generation, privacy attacks, and defenses against such attacks, as well as the use of TEEs in encrypted deduplication.

# 3

# Related Work

## Contents

In this chapter, we present the related work in the areas of encrypted deduplication, key generation, privacy attacks and defences against such attacks, as well as the use of TEEs in encrypted deduplication.

The chapter is organized as follows: Section 3.1 presents MLE, an approach to key generation which derives the key from the data content. Section 3.2 presents Duplicateless Encryption for Simple Storage (DUPLESS), a system which employs a dedicated key server for encrypted deduplication. Section 3.3 presents the use of MinHash in encrypted deduplication to protect against frequency analysis attacks. Section 3.4 presents a system that enables tunable-encrypted deduplication, specifying the trade-off between storage efficiency and data confidentiality. Section 3.5 presents a solution that uses TEEs to offload expensive cryptographic operations and improve performance in encrypted deduplication. Finally, Section 3.6 presents a system that uses TEEs and keeps a frequency table inside the enclaves in order to hide this information from untrusted components.

## 3.1 MLE: Message Locking Encryption

Some encrypted deduplication approaches preserve the deduplication capability by deriving the key for encryption and decryption from the chunk content, usually via the hash or fingerprint of a given chunk. MLE [13] is an example of this approach.

The key generation algorithm in an MLE scheme maps a message *M* to a key *K*. The encryption algorithm takes two inputs, a key *K* and a message *M*, and produces as output a ciphertext *C*. The decryption algorithm takes as input a key *K*, and a ciphertext *C*, and produces a plaintext *M*, allowing for the recovery of the original plaintext. MLE also defines a tagging algorithm that maps a ciphertext *C* to a tag *T*, this tag simplifies the identification of duplicates. If *M1* and *M2* are the same, their ciphertext will be the same, and the same tag will be generated. It is then possible to use the tags to detect duplicates, rather than comparing the entire ciphertext.

An important property of MLE is Tag Consistency (TC), which ensures the integrity of the stored ciphertext (meaning that an attacker cannot make an honest client download and recover a message different than the one that the client stored encrypted). Some MLE schemes provide a strictly stronger property, named Strong Tag Consistency (STC), that makes additionally hard to create $(M, C)$ such that $T(C) = T(SKE(K(M), M))$ but $D(K(M), C) = \perp$, which prevents an attacker from erasing an honest client's message.

A trivial form of achieving MLE is by letting the key *K* be equal to the message *M*. However, this solution does not have storage advantages since the client must store as a key the entire file, resulting in no storage savings. A requirement in MLE is that keys must be shorter than messages, and ideally, these should have a fixed, short length. The are several variants of MLE. Most are based on the use of

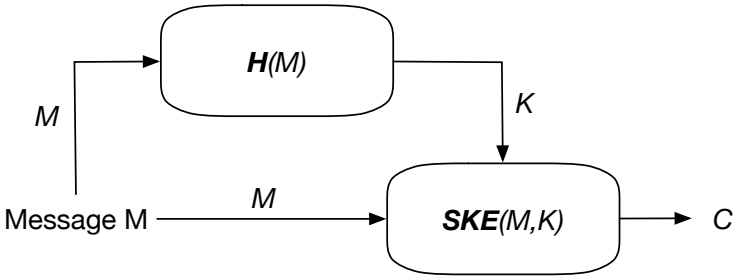


**Figure 3.1:** Convergent Encryption.

a hash function *H* and an Symmetric-Key Encryption (SKE) scheme.

The first major variant is Convergent Encryption (CE) [13], illustrated in Figure 3.1, in which the key is the hash of the message *M*, the ciphertext is the encryption of *M* using *K* using an SKE scheme, and finally, tag *T*, which is generated as the hash of the ciphertext:

$$K = \mathbf{H}(M) \tag{3.1}$$

$$C = \mathbf{SKE}(K, M) \tag{3.2}$$

$$T = \mathbf{H}(C) \tag{3.3}$$

Another major variant is Hash and Convergent Encryption 1 (HCE1) [13]. HCE1 still generates the key *K* as the hash of *M*, as seen in Equation 3.1, but it generates ciphertext *C* as the concatenation (|| is the concatenation operator) of the encryption of *M* and the tag *T*, generated as the hash of *K*:

$$C = \mathbf{SKE}(K, M) \,||\, T \tag{3.4}$$

$$T = \mathbf{H}(K) \tag{3.5}$$

The reasoning behind HCE1 is to provide better performance for the server, which can retrieve the tag *T* from the second part of the ciphertext, so it does not need to compute it by hashing the ciphertext, which can be possibly long. Unfortunately, HCE1 is vulnerable to duplicate faking attacks and cannot ensure TC.

A duplicate faking attack is one in which a legitimate message is undetectably replaced by a fake one, for example, when a client has already stored *C*, with a generated tag *T*, and later an attacker sends *C´* with the same tag *T*, replacing the original message.

In order to achieve better performance, while still providing consistency, two new schemes were introduced, namely Hash and Convergent Encryption 2 (HCE2) [13] and Randomized Convergent Encryption (RCE) [13].

HCE2 is just as efficient as HCE1, it requires two passes through the data, one for generating the key, and a second one for encryption. HCE2 modifies HCE1 to directly include a mechanism called protection decryption, which helps it achieve TC security. The decryption routine now also checks for tags embedded in the ciphertext by recomputing the tags using the just decrypted message

RCE is even more efficient since it only requires a single pass through the data in order to generate the key, encrypt the data, and produce the tag. RCE is able to achieve such high performance by using a random scheme, while CE, HCE1, and HCE2 were deterministic.

The RCE encryption scheme first picks a fresh random key *L* and computes $C = SKE(L, M)$ and $K = H(M)$ in the same pass, finally encrypts *L* with *K* as a one-time pad, together with tag *T*, which is generated in the same way as in CE:

$$L = \textbf{RAND}() \tag{3.6}$$

$$C = \textbf{SKE}(L, M) \tag{3.7}$$

$$K = \textbf{H}(M) \tag{3.8}$$

$$C' = C \,||\, K \oplus L \,||\, T \tag{3.9}$$

Table 3.1 shows the comparison between the multiple schemes of MLE.

**Table 3.1:** MLE Schemes Comparison.

| Scheme | Speed | Integrity | |
|---|---|---|---|
| | | TC | STC |
| CE | Slow | yes | yes |
| HCE1 | Fast | no | no |
| HCE2 | Fast | yes | no |
| RCE | Faster | yes | no |

HCE1 does not provide tag consistency. The good news is that CE, HCE2, and RCE all implement TC security, so an attacker can't get the client to recover a different file than what he uploaded. But only CE provides STC security, which means that the server cost reduction provided by HCE1, HCE2 and RCE comes at a price, i.e. loss of STC security. This lets us conclude that there is a trade-off between performance and integrity. If we wish to have the best performance possible and still provide TC, then using RCE is the best option. If we wish to provide the integrity properties (TC and STC), then the only option is to use CE.

## 3.2 DupLESS: Duplicateless Encryption for Simple Storage

CE allows us to store encrypted data, with integrity properties, while still allowing deduplication. However, as discussed before, CE is subject to an inherent security limitation, namely, the susceptibility to brute force attacks [15].

DUPLESS [15] is a system that strengthens the security of encrypted deduplication against offline brute force attacks. It deploys a dedicated Key Server (KS) for MLE key generation, separate from the SS. Deduplication is performed at file level instead of chunk level. DUPLESS also introduces deduplication *heuristics*. They are used to determine whether a file that is about to be stored on the SS should be selected for deduplication, or processed using a randomly generated key. One example can be very small files or sensitive files that can be prevented from being deduplicated.

DUPLESS was designed such that it can be integrated into existing systems, such as Dropbox [1] and Google Drive [2], as illustrated in Figure 3.2.

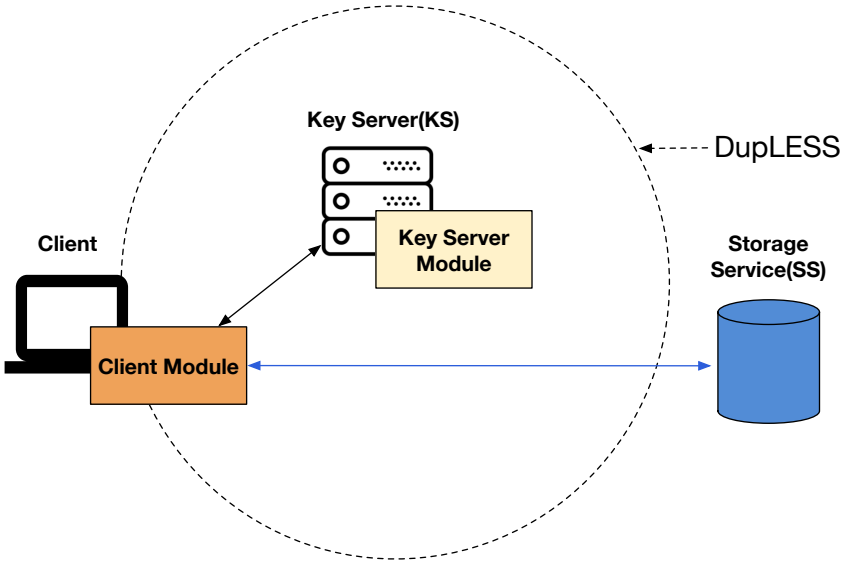


**Figure 3.2:** DUPLESS System Design.

To encrypt a plaintext chunk, a client first sends the fingerprint of the plaintext chunk to the key server, which returns the MLE key via both the fingerprint and a global secret maintained only by the key server:

$$K = \mathbf{H}(k \,||\, P) \tag{3.10}$$

where *k* is the global secret owned by the KS and *P* is the fingerprint of a given chunk. This increases the security of MLE and still retains the deduplication properties.

The advantage of having a dedicated KS is to have a main line of defense against different types of attacks. If we want to deal with external attackers, such as a compromised SS, a possible solution could

be that the KS should authenticate the clients with whom it is interacting. If the attacker is more powerful and resourceful and is able to compromise a client, then we are dealing with brute force attacks from this client, but unlike CE, here the offline brute force attacks are online, they have to involve the KS, they are going to be much slower and easier to detect.

In face of a severe attack, such as the KS being compromised, the attacker could be able to retrieve all the keys for all ciphertexts generated on the KS. DUPLESS is not vulnerable to this problem since it uses a Oblivious Pseudorandom Function (OPRF) [22] protocol based on RSA blind-signatures [23–25] when communicating with clients; the KS learns nothing about client input, and the client only learns *K*.

An OPRF protocol is something that stands between a client and a server, as shown in Figure 3.3, where *f* is the original input, for example, the chunk data. The server holds a global secret, and the objective of the client is to receive the output of the OPRF, at the end of the communication, the client must not know anything other than the output, and the server should not know any of the inputs given by the client.

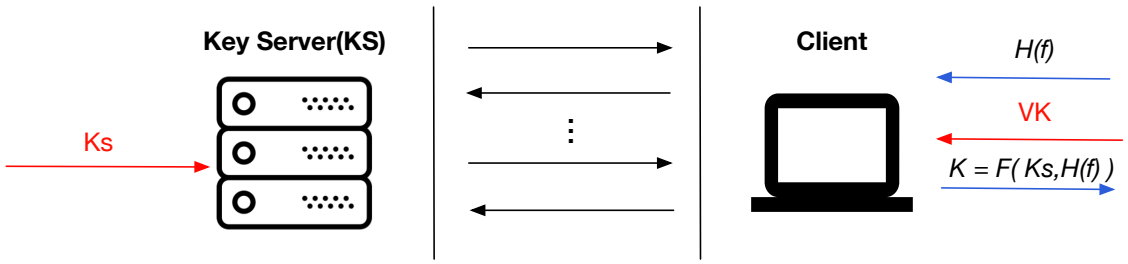


**Figure 3.3:** OPRF Protocol.

If the KS is down, overloaded, or subjected to a denial-of-service attack, encryption resorts to SKE, generating a random key as the encryption key. This ensures availability, but there is no deduplication during that period of time.

The significant increase in security comes at the cost of a moderate price in terms of performance and a small increase in storage requirements relative to the base system. The low-performance overhead is partly due to the optimization of the client-to-KS OPRF protocol, but also to ensure that DUPLESS uses fewer interactions with SS.

In short, DUPLESS is a system that provides encrypted deduplication with the aid of a KS. It was the first solution to provide secure deduplication without compromising resilience.

## 3.3 MinHash Encryption

Most state-of-the-art encrypted deduplication systems adopt a deterministic encryption approach to encrypt each plaintext chunk with a key derived from the content of the chunk itself. However, such

a deterministic approach reveals the underlying frequency distribution of the original plaintext chunks, allowing attackers to launch frequency analysis attacks against the resulting ciphertexts, and infer the content of the original plaintext chunk. MLE suffers from another security issue, which is that it cannot fully protect against content leakage because the encryption approach is deterministic.

To resist frequency analysis, the idea is to disturb the frequency ordering of ciphertext chunks. To this end, it was considered an encryption scheme called MinHash encryption, which derives an encryption key based on the smallest fingerprint on a set of adjacent chunks, so that some identical plaintext chunks can be encrypted into multiple different ciphertext chunks.

MinHash encryption works as follows. First, it groups multiple consecutive plaintext chunks into segments. We can view a segment as a set of chunks.

For each segment, it derives a key as the minimum fingerprint of all chunks in the segment, for example, by comparing only the less significant bits. Then it encrypts all the chunks present in the segment using the same key. In backup workloads, the segments are often similar with a large fraction of duplicated plaintext chunks, so the keys or minimum fingerprints that will be generated for similar segments are likely to be the same. In this way, most duplicated chunks are encrypted with the same key, making deduplication viable after encryption.

Regarding security advantages, MinHash encryption has been shown to effectively reduce the overhead of server-aided MLE since it sends only as many fingerprints as the number of segments to the KS, increasing bandwidth efficiency. It can also be used to break the deterministic nature of encrypted deduplication and disrupt the frequency ranking of ciphertext chunks.

MinHash is robust against the locality-based attack, by breaking the deterministic nature of encrypted deduplication.

Still, there are deduplication issues when using MinHash encryption. Some of the same plaintext chunks may still reside in different segments, with different minimum fingerprints and different keys, so the ciphertext chunks they generate will be different and cannot be deduplicated, resulting in a slight decrease in storage efficiency.

However, such near-exact data deduplication is enough to change the overall frequency ranking of the ciphertext chunk by using different keys to encrypt a small part of the repeated chunks, thereby invalidating the frequency analysis.

Additional issues have been raised in other works [3], namely that MinHash provides limited protection and limited configurability. As for limited protection, MinHash encryption is based on the assumption of file similarity to ensure its deduplication effectiveness, so its storage efficiency may not be suitable for general workloads. More importantly, the randomness of the smallest chunk fingerprint in the segment is limited (otherwise the deduplication effect will be lost), so MinHash encryption only slightly undermines the certainty of MLE, and does not provide security guarantees for frequency analysis.

For limited configurability, MinHash does not provide a configurable mechanism to quantify the trade-off between storage efficiency and data confidentiality. MinHash encryption disrupts the frequency ordering of ciphertext chunks by sacrificing storage efficiency (for example, repeated plaintext chunks in different segments are encrypted with different keys and cannot be deduplicated after encryption).

## 3.4 TED: Tunable Encrypted Deduplication

Tunable Encrypted Deduplication (TED) [3] appeared to solve the issues presented at the end of the previous section. TED is a cryptographic primitive that provides an adjustable mechanism that allows users to balance storage efficiency and data confidentiality.

Like in DUPLESS [15], TED uses a dedicated KS, which is responsible for key generation, and a SS responsible for performing deduplication. The only fully trusted component is the KS. The justification for this assumption is that the KS is deployed by companies or individuals that outsource deduplication. The SS can be external, which means that it is considered trustworthy but curious.

TED is an encrypted deduplication primitive that aims to achieve *Configurability*, that is, quantify the trade-off between storage efficiency and confidentiality of data such that information leakage is minimized. Also, TED keeps a record, Count-Min Sketch (CM-SKETCH) [26], of the frequency of each chunk. There are two advantages of using CM-SKETCH. First, it limits the amount of memory used to track the frequency of all chunks and errors are bounded. Second, the approximate count protects chunk information from being affected by the KS, which is a security requirement in DUPLESS [15].

The principle behind TED is to derive the key of each plaintext chunk based on two additional inputs: its *current frequency f*, that is, the number of duplicate copies of *M* that have been uploaded by all clients; and the *balance parameter t*, which controls the trade-off between storage efficiency and data confidentiality. The key *K* for *M* is generated by the KS in the following way:

$$K = \mathbf{H}(k \,||\, P \,||\, f/t) \tag{3.11}$$

where *k* is the global secret owned by the key manager, *P* is the chunk fingerprint, and $f/t$ is the maximum integer smaller than *f/t*.

*f* is a cumulative function, which means that it increases as more copies of the same data are uploaded, meaning that key *K* will be updated as the value of $f/t$ increases.

Therefore, according to the value of *t*, copies of *M* will generally be encrypted with different keys. If *t* = 1, each copy of *M* has a different *K* and TED is reduced to SKE; if $t \to \infty$, all duplicates of *M* have the same *K*, and TED is reduced to MLE. Intuitively, *t* can be seen as the maximum number of duplicate copies of a ciphertext chunk.

To upload a file in TED (see Figure 3.4), the client divides the file data into chunks, generates a key

for each chunk through interaction with the KS, encrypts each chunk with the corresponding key, and then uploads the chunk to the SS. Furthermore, for file reconstruction, the client generates a *file recipe*, which lists the chunk fingerprint and chunk size according to the chunk order in the file, as well as a *key recipe* that retains the keys of all chunks. It uses the master key of each client to encrypt the *file recipe* and the *key recipe* for protection and uploads them to the SS together with the ciphertext chunk. The SS performs data deduplication on the ciphertext chunk. It maintains a *fingerprint index*, which is a key-value store, used to track fingerprints of physical chunks for duplicate detection. The SS does not apply deduplication to metadata, instead, it directly saves the *file recipe* and *key recipe* (in encrypted form) in physical storage.

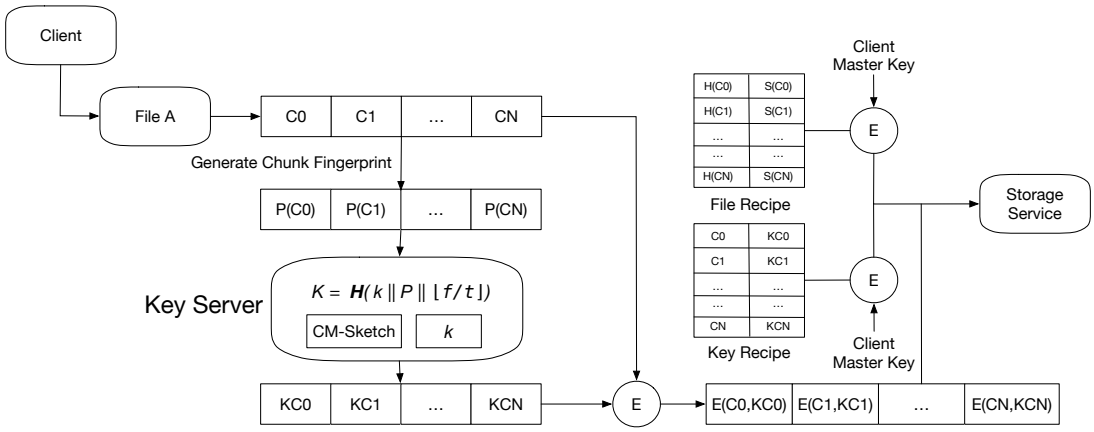


**Figure 3.4:** TED File Upload.

To download a file (Figure 3.5), the client first retrieves the *file recipe* and the *key recipe* from the SS, and uses its master key to decrypt them. Then it retrieves the ciphertext chunks from the SS according to the *file recipe* and decrypts them with the key stored in the *key recipe*.

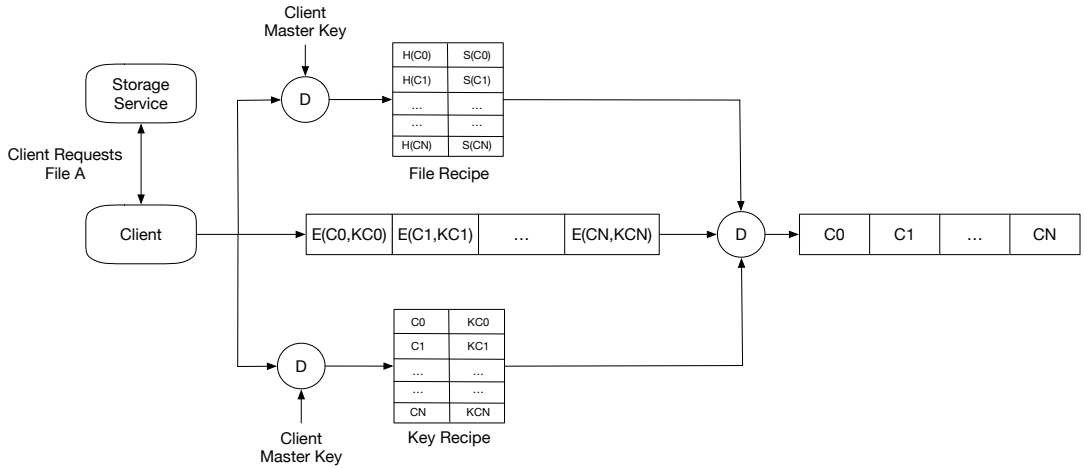


**Figure 3.5:** TED File Download.

The key generation scheme in Equation 3.11 raises a security problem. For the same file with the same chunk sequence, Equation 3.11 will return the same keys, which also result in the same ciphertext chunk sequence, allowing the attacker to infer whether the two encrypted files are initially the same. Therefore, the key generation must generate different ciphertext chunk sequences for the same file while maintaining the effectiveness of deduplication.

To solve this issue, TED implements a probabilistic key generation method that can non-deterministically encrypt the same file (with the same plaintext chunk sequence) into different ciphertext chunk sequences, while maintaining the effectiveness of deduplication.

The idea of TED is to randomly select a chunk's key from a set of candidates, instead of returning the same key as in Equation 3.11. Specifically, for each plaintext chunk *M*, let *x* = $f/t$, where *f* is the current frequency of *M*, and *t* is the balance parameter. After receiving the hash value of *M*, the key manager calculates a key seed candidate $k_x$ as:

$$k_x = \mathbf{H}(k \,||\, h_1(M) \,||\, h_2(M) \,||\, ... \,||\, h_r(M) \,||\, x) \tag{3.12}$$

Then a key seed is uniformly selected from the candidate set $\{k_0, k_1, ..., k_x\}$:

$$k \xleftarrow{\$} \{k_0, k_1, ..., k_{x-1}, k_x\} \tag{3.13}$$

The client can then derive the key of *M* as

$$K = \mathbf{H}(k \,||\, P) \tag{3.14}$$

where *P* is the fingerprint of M. TED did not use *k* as the key of *M* to prevent the key manager and attackers who can eavesdrop on the key manager's reply from directly accessing the key.

As we observe more duplicates of *M* (that is, an increasing *f*), the most recent duplicates of *M* are encrypted based on some old key seeds in $\{k_0, k_1, ..., k_x\}$ used before. Therefore, TED maintains the effectiveness of deduplication by allowing the same key seed to protect some duplicates. At the same time, the generation of ciphertext chunks is uncertain because they come from a randomly selected key seed (as opposed to the deterministic key generation in Equation 3.11).

Generally speaking, a plaintext chunk with a higher frequency will be encrypted into a more diverse set of ciphertext chunks, because more candidate key seeds can be selected as *f* increases.

## 3.5 SGXDedup: Accelerating Encrypted Deduplication via SGX

Encrypted deduplication retains the effectiveness of deduplication on encrypted data and is attractive to outsourced storage. However, existing encrypted deduplication methods are based on expensive

encryption primitives, which can cause a significant performance drop.

Server-assisted key management (such as DUPLESS [15] and TED [3]) requires expensive encryption operations [22] to prevent the KS from knowing the plaintext chunk and key during key generation.

Intel SGX provides a type of TEE, called an enclave, that allows data processing and storage with confidentiality and integrity [27]. The expensive cryptographic operations of encrypted deduplication can be offloaded by directly running sensitive operations in enclaves, thus improving the performance of encrypted deduplication while maintaining its security, bandwidth efficiency, and storage efficiency.

SGXDedup [28] is a high-performance SGX-based encrypted data deduplication system. SGXDedup is built on server-aided key management such as DUPLESS [15], also using MLE key generation, but performs efficient cryptographic operations inside enclaves. It uses source-based deduplication and Proof-of-ownership (POW) [29]. POW is an encryption method that enhances source-based deduplication to prevent side-channel attacks while maintaining the bandwidth savings of source-based deduplication.

The idea is to let the SS verify that the client is indeed the owner of the ciphertext chunk and is authorized to have full access to the ciphertext chunk. This ensures that a malicious client cannot query the existence of other clients' chunks. Specifically, in POW-based source-based deduplication, the client attaches a POW certificate to each fingerprint sent to the SS, and the SS can use it to verify whether the client is the true owner of the corresponding ciphertext chunk. The SS-only response after successful proof verification prevents any malicious client from recognizing ciphertext chunks owned by other clients.

Figure 3.6 presents the architecture of SGXDedup, which has multiple clients, a KS, a SS and two enclaves: the *key enclave* and the *POW enclave*.

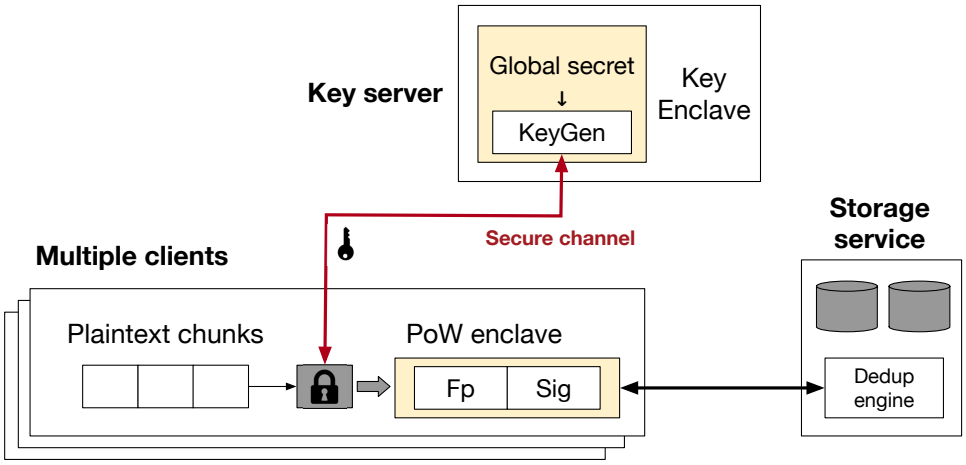


**Figure 3.6:** SGXDedup Architecture.

SGXDedup deploys a key enclave in the KS to manage and protect the global secret of server-aid

MLE [13], preventing a malicious or compromised KS from leaking the secret.

In order to perform MLE key generation, the key enclave and the client first establish a secure channel based on the shared blinded key, after that the client submits the fingerprint of the plaintext chunk through the secure channel. The key enclave calculates the MLE key as a cryptographic hash of the global secret and fingerprint:

$$K = \mathbf{H}(k \,||\, P) \tag{3.15}$$

It then returns the MLE key through a secure channel. The key enclave benefits both performance and security. It protects fingerprints and MLE keys through a secure channel based on a shared blinded key so that the key server cannot learn any information from the MLE key generation process.

SGXDedup deploys a POW enclave on each client to prove the authenticity of the ciphertext chunk in source-based deduplication. The POW enclave first establishes a shared POW key with the SS. SGXDedup uses Diffie-Hellman Key Exchange (DHKE) to implement the key agreement After the POW key is generated, the client encrypts each plaintext chunk into a ciphertext chunk.

The POW enclave takes the ciphertext chunk as input, calculates the corresponding fingerprint, and uses the POW key shared with the SS to create a fingerprint signature. Then the client uploads the fingerprint and signature to the SS. The SS verifies the authenticity of the fingerprint based on the corresponding signature and POW key.

Only when the fingerprint is authenticated, the SS will continue to check whether the fingerprint corresponds to any duplicate ciphertext chunks that have been stored. The client verifies the ownership of the ciphertext chunk, not the ownership of the plaintext, to protect the original information from the SS.

SGXDedup has advantages over previous solutions [15] [3] [17], it has better bandwidth efficiency due to the use of source-based deduplication, it has better key generation performance due to the fact that blind fingerprints (OPRF protocol) are replaced with ECALLs to enclaves, and better security of secret storage, since secrets are stored inside enclaves.

However, SGXDedup does not provide a mechanism to quantify the trade-off between storage efficiency and data confidentiality, meaning that the same plaintext will always map to the same ciphertext, making it vulnerable to frequency analysis attacks. Moreover, since it uses source-based deduplication, the client only uploads the non-duplicated ciphertext chunks to the cloud, meaning that it is able to detect if deduplication occurred (Deduplication Detection).

## 3.6 S2Dedup: SGX-Enabled Secure Deduplication

SGX Enabled Secure Deduplication (S2DEDUP) [30] is a trusted hardware-based privacy deduplication system designed to support multiple security schemes that allow different levels of performance, security

assurance and storage savings. S2DEDUP leverages Intel SGX to enable cross-user deduplication at third-party SSs.

As previously said, S2DEDUP supports multiple secure schemes, these are:

- Simple - Basic secure scheme, a chunk is ciphered with the enclave key;
- Epoch - Similar to the Simple scheme, however, deduplication is performed in epochs (e.g., a predefined number of operations or a period of time), at the end of an epoch a new random key is generated in order to calculate the hash of new chunks;
- Estimated - This scheme defines an upper bound for the number of duplicated copies that each stored chunk may have. This solution uses the CM-SKETCH [26] algorithm, which is used to deduce the approximated number of copies per stored chunk;
- Exact - Combines the security advantages of the previously 2 presented schemes (Epoch and Estimated). The epoch-based approach sets a time limit for duplicate detection, while the frequency-based approach allows masking the number of duplicates found in an epoch. This scheme, instead of relying on an estimated counter, keeps an in-memory hash table in the enclave, that maps a hash to its frequency. When this hash table reaches the enclave memory limits, an epoch is reached and the table is reset to its original state.

To send a file to the storage system, the client will divide the file into chunks and encrypt each chunk with a symmetric key (client key) that only the client and enclave know, the encrypted chunks are sent over the network using the Internet Small Computer System Interface (ISCSI) protocol, where each request can contain several chunks of fixed size. On the server, the deduplication engine parses the data from the client request, chunk by chunk to detect duplicates. The hash of each chunk is compared with the hashes of the stored chunks. The enclave is responsible for the hash calculation, as the chunks need to be decrypted with the client key, and encrypted with the universal key. There is also an index table that maps the hash of a chunk to a structure that holds the encrypted chunk's physical address, and the number of logical addresses that reference this physical address. A chunk $j$ was previously stored if there is an entry with its hash in the index table.

The client chunks are decrypted with the client key in the enclave and are encrypted with a universal key, to obtain secure deduplication. Obtaining the hash and encrypting the chunks is done in different steps since it is only necessary to encrypt chunks when their hash is not present in the index table.

When a client wishes to retrieve data from the SS, the client sends their logical addresses, then, the metadata component is queried to get the chunk's physical addresses. The chunk present in each physical address is deciphered with the universal key and encrypted with the client key (this is done at the enclave). The chunks are then sent to the client, where they will be decrypted using the client key.

S2DEDUP [30] solves the deduplication detection issue that SGXDedup has by performing deduplication detection on the server-side, it performs key-generation based on the frequency of chunks in order to protect against frequency analysis, however, there are a few limitations to this solution.

One of the limitations of S2DEDUP is that, due to the limited memory of the enclave (128MB) [31], it is not possible to store the frequency of all chunks. Because of this, it may be necessary to reset the table multiple times, changing the keys for encryption more times than necessary, limiting the deduplication effectiveness.

Another limitation is the intervention of the enclave in read operations, every time a client requires a chunk the enclave must decipher the chunk with the universal key, and cipher it with the client key. This creates a bottleneck in the system since the number of enclaves is limited by the number of processors. Also, the enclave must keep in memory the keys for all clients, which creates a storage issue.

## 3.7 Discussion

In this section, we discuss and compare the previously presented work, summarize the tradeoffs that each solution offers, and how their mechanisms can assist in achieving our goals. Table 3.2 presents the main aspects and mechanisms of each solution.

**Table 3.2:** Comparison of the main systems and approaches to deduplication.

| System / Approach | Deduplication Approach | Crypto Approach to Avoid Key Server Analysis | Tunable | Key Generation Approach |
|---|---|---|---|---|
| MLE | Target and Source Based | - | No | CE / HCE1 / HCE2 / RCE |
| DupLESS | Target-based | Blinded Fingerprint | No | Convergent Encryption(MLE) |
| MinHash | Target-based | Minimum Fingerprint of Chunks in Segment | No | MinHash Encryption |
| TED | Target-based | Blinded Fingerprint | Yes | Probabilistic Key Generation |
| SGXDedup | Source-based | Enclaves | No | Convergent Encryption(MLE) |
| S2Dedup | Target-based | Enclaves | Yes | Probabilistic Key Generation |

Deduplication systems and approaches that offer target-based deduplication such as DUPLESS, TED, MinHash and S2DEDUP incur in extra communication load for uploading all the data to the server. On the contrary, source-based deduplication systems such as SGXDedup are intrinsically vulnerable to deduplication detection from the client side, despite the use of the enclave.

There is a visible trend in the related work on requiring a key server as a trusted third party (mediator) to support cross-user deduplication by centralizing the key generation. This solution followed by most systems [3, 15, 28] requires the use of heavy cryptographic operations to avoid revealing sensitive information, such as the chunk content, to the key server. DUPLESS and TED use the OPRF protocol [22], which can impose a significant performance overhead. Additionally, constant communication with the KS for key generation of each chunk can add significant latency and turn the KS into a bottleneck. In

Li *et al.* [17] they send the minimum fingerprint of the segment, as this depends on all the chunks fingerprints inside the segment, SGXDedup guarantees that the KS can't record any information since all the interactions are done between the client and enclave, and S2Dedup also guarantees that the server can't observe what's being passed to the enclave since the data is ciphered by the client.

As for quantifying the trade-off between storage efficiency and data confidentiality, only TED and S2DEDUP provide a tunable mechanism, while other solutions [13, 15, 17, 28] do not offer such configurability.

The final relevant characteristic of these systems is their key generation algorithm. Both DUPLESS and SGXDedup rely on CE, the most secure variant of MLE described in Section 3.1, where each chunk will always map to the same key. MinHash also implements a deterministic key generation mechanism, however, the key depends not only on the chunk but on the whole segment, since the key for encryption is generated based on the minimum fingerprint of the segment chunks. Unfortunately, the use of deterministic mechanisms for key generation leaves these systems vulnerable to frequency analysis attacks. Finally, TED and S2DEDUP use probabilistic key generation, meaning that a chunk may map to different keys with the progression of time, hiding possible frequency patterns in the stored data.

The related work aims to design storage services that offer deduplication while protecting the data content and achieving high performance, in Table 3.3 we compare these systems against the possible attacks and performance metrics, described in Section 2.3.

Also, we add an entry for what FH-Dedup provides when compared to these systems.

**Table 3.3:** Comparison of protection against attacks and performance metrics in different systems and approaches, where N is the number of chunks.

| System / Approach | Protection against brute force attacks | Protection against deduplication detection | Protection against frequency analysis | Frequency Storage | Key Generation Communication Protocol | Number of Round Trips | Mediator Intervention in Read Operations |
|---|---|---|---|---|---|---|---|
| MLE | No | No | No | N/A | - | 2 * N | No |
| DupLESS | Yes | Yes | No | N/A | OPRF | 2 * N | No |
| MinHash | Yes | Yes | No | N/A | - | 2 * N | No |
| TED | Yes | Yes | Yes, to some degree | Full | OPRF | 2 * N | No |
| SGXDedup | Yes | No | No | N/A | ECALLS / OCALLS | 2 * N | No |
| S2Dedup (Exact) | Yes | Yes | Yes, to some degree | Epoch-Based | ECALLS / OCALLS | N | Yes |
| FH-Dedup | Yes | Yes | Yes, to some degree | Full | ECALLS / OCALLS | N | No |

All previous solutions [3, 15, 17, 28] can prevent brute-force attacks, except MLE, because all these systems and approaches use a dedicated key mediator responsible for key generation and distribution. For this reason, all clients must interact with the mediator before encrypting chunks and sending them to the SS. FH-Dedup uses the same technique to protect itself from brute force attacks.

Against deduplication detection attacks, DUPLESS, TED, MinHash and S2DEDUP use target-based deduplication to protect from this attack. This means that the SS is responsible for performing deduplication on ciphertext chunks. However, SGXDedup does not have full protection against this attack, since it uses source-based deduplication. A malicious client can always observe side channels to detect if the enclave (on the client side) uploads or performs deduplication on the desired chunk. The client can use

side channels, such as measuring the network bandwidth or the response time from the enclave, and others. FH-Dedup does not allow someone to detect if deduplication occurred, this is done through the use of target-based deduplication.

Probabilistic key generation ensures that only a fixed number of plaintext chunks will be encrypted using the same key, and posterior identical plaintext chunks will use different keys. This approach diminishes storage savings but improves security against frequency analysis, which is an acceptable trade-off. For this reason, FH-Dedup shall use the same key generation as TED and S2DEDUP, protecting itself against frequency analysis. Also, we want to maximize storage savings by using deduplication, in conjunction with such mechanisms to protect against frequency analysis, S2DEDUP loses its frequency record every time the enclave memory reaches its maximum, losing the deduplication effect. Our system has the same storage savings as TED.

Performance during key generation is also an important aspect, as stated above, DUPLESS and TED use the OPRF protocol to avoid input analysis from the KS, which is a heavy operation. Since this operation is costly, solutions such as SGXDedup and S2DEDUP, which use enclave ECALLS/OCALLS to perform sensitive cryptographic operations (e.g., key generation), will have better performance, since it does not require heavy operations to hide client input. FH-Dedup achieves good key generation performance as SGXDedup and S2DEDUP through the use of TEEs.

All previous systems (besides S2DEDUP) require 2 * *N* round trips to store information in the SS, where N is the number of chunks. This is because the keys for encryption must be obtained by interacting with the KS, and later, all encrypted chunks must be sent to the SS. Even though solutions such as SGXDedup query the SS to check what chunks it must send, it incurs in extra round trips to retrieve that information. Our system minimizes the number of round trips to *N*.

Finally, the performance while retrieving data is also important, S2DEDUP requires the intervention of the enclave to re-encrypt chunks, this can be costly if the number of chunks present in a file is high. Our FH-Dedup system does not require interaction with the enclave in read operations, this way we achieve higher throughput.

## Summary

In this chapter, we introduced the most relevant systems and approaches to encrypted deduplication, MLE, DUPLESS, MinHash, TED, SGXDedup and S2DEDUP. We discuss their advantages, and disadvantages, more precisely that there is a noticeable trade-off between storage efficiency and privacy.

In the next chapter, we will discuss our system architecture. We begin with the trust assumptions, then present the various system components and the interactions between those components. Finally, we present a criterion that we will use to assess the privacy guarantees offered by FH-Dedup.

# 4

# FH-Dedup

## Contents

This chapter describes the architecture and implementation of FH-Dedup. Our system is inspired by previous work and is based on the realization of a secure mediator, running in a server enclave.

## 4.1 Trust Assumptions

We make the following assumptions:

1. The integrity and confidentiality of the data sent through the communication channel between the client and server are protected by the use of cryptographic operations;

2. Communication between the client and the enclave is secure;

3. There is no collusion between the client and the storage server;

4. The recipe files (explained in more detail ahead) are stored securely, as they are protected with the client's key;

5. The enclave guarantees confidentiality and integrity of the code and data held therein.

We consider an honest but curious attacker, that is, one who does not change the system protocol but who has access to a set of auxiliary data and observes the access frequency distribution of the hashes of the chunks.

This attacker aims to identify the original content of encrypted chunks on the server by observing the accesses in the server (comparing the frequency of accesses of a given entry to the frequency of a given chunk hash), the accesses to the data itself, and the accesses to the metadata of the system, specifically the entry point for the encrypted table stored in the untrusted area on the server (presented in the following paragraphs).

## 4.2 Components and Interactions

The architecture of FH-Dedup can be seen in Figure 4.1. It has 3 main components: *Clients*, which hold the chunks (in plaintext) that will be later sent and stored in the server. The *Server* (or SS), which contain the encrypted data of the clients, and the *Enclave* that runs in the server.

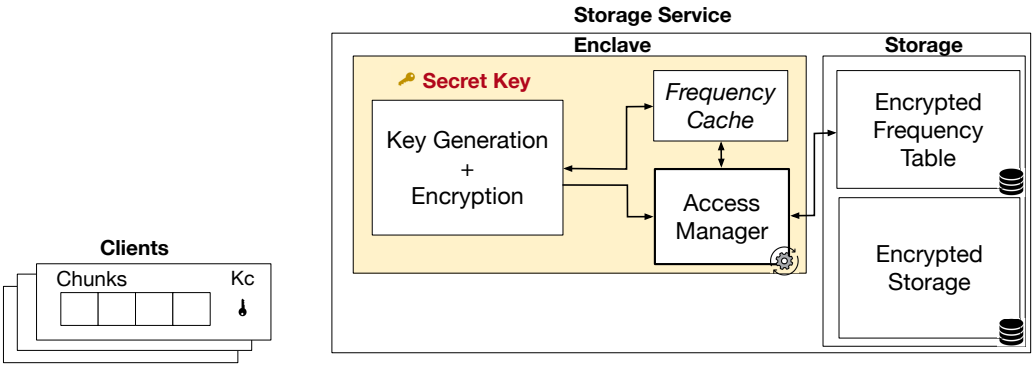


**Figure 4.1:** FH-Dedup Architecture.

Similarly to the related work, we consider that clients can write and read their data as they wish. This is done by establishing a secure connection with the enclave present on the SS, which is responsible for encrypting the data and placing it in encrypted storage. The enclave will choose the key to encrypt the data, taking into account the frequency of each of the chunks. Next, we describe the operations of sending and reading a file on FH-Dedup.

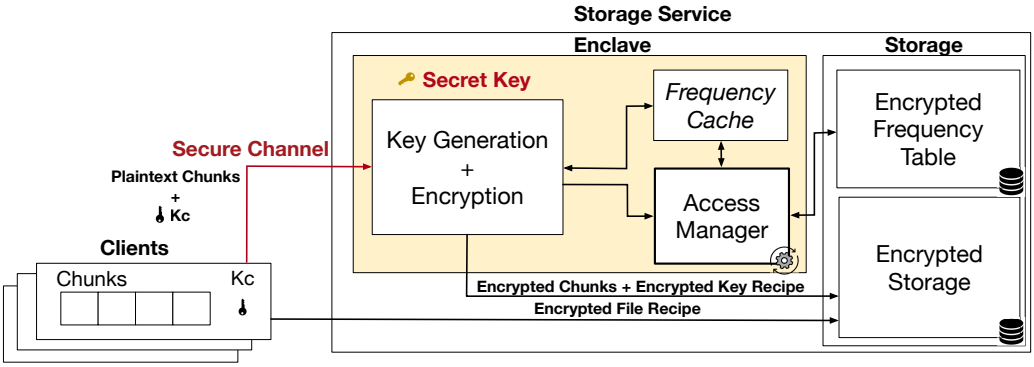


**Figure 4.2:** Write/Send data operation.

The write/send operation is summarized in Figure 4.2. A client who wants to upload a file *F*, first breaks the file into multiple chunks. It also generates a file recipe, which lists the chunk hash and the chunk size based on the chunk order in the file. The size of the chunks is a system-wide constant, e.g., 4 KB. The client then encrypts the file recipe with a client key *Kc* (unique for each client) and sends it to the SS. The client then sends all chunks and the client key to the encryption enclave. The enclave will then generate a key for each chunk based on its frequency, in a parameter *t* that indicates the maximum number of equal copies that can be encrypted with the same key, and a secret known only to the enclave, and encrypt each chunk with the respective key. After generating all cryptograms, the enclave builds a key recipe, which contains the keys for all the chunks that were encrypted. The enclave encrypts this recipe with the client key *Kc*, and stores the encrypted chunks and the encrypted recipe file in the SS.

To read a file (see Figure 4.3), a client retrieves the encrypted file recipe and the encrypted key recipe from the SS, as well as the encrypted chunks. Then it decrypts the recipe files with the *Kc* client key. Finally, the client decrypts each encrypted chunk using the respective key and reconstructs the file based on the file recipe.

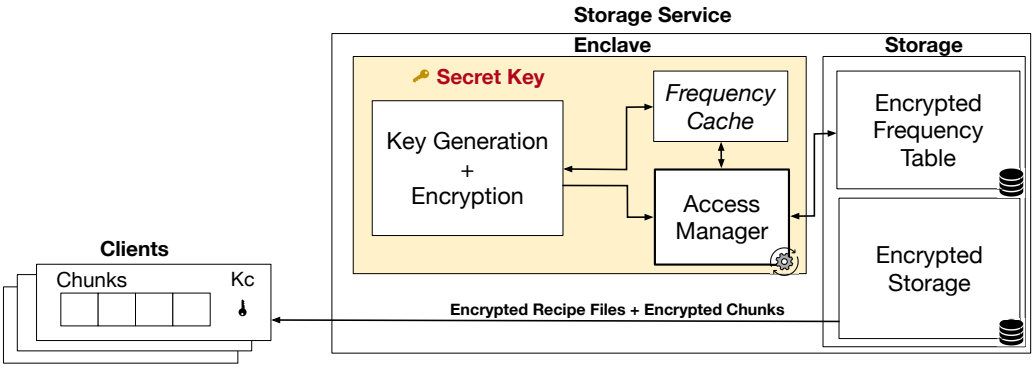


**Figure 4.3:** Read data operation.

Our system improves on previous solutions [15] [3] [28] by only needing one single interaction with the encryption enclave, whereas in previous work they would have as many interactions with the KS as the number of chunks, to obtain the key for each chunk.

It also removes the need to perform OPRF [22] blind fingerprints to hide input information from the key server as previous solutions [15] [3] did, since we use an enclave to perform key generation and encryption, we trust the encryption enclave and the code running on it, so we know that it is not recording information about user input. Also, in read operations there is no interaction with the enclave in any part of the process since clients retrieve information from the SS and reconstruct the file themselves, as opposed to S2DEDUP [30], which always contacts the enclave to do re-encryption.

Our system resists brute force attacks by utilizing an encryption enclave as its key mediator. This means that the SS must communicate with the key mediator during any attack. This delays the time of attack due to the SS having to communicate with the key mediator.

As for deduplication detection attacks, we use target-based deduplication, the client always sends the plaintext chunks to the encryption enclave. Also, to avoid any sort of analysis (time or network) done by an attacker, the encryption enclave always performs encryption on all plaintext chunks. After encrypting all plaintext chunks, it will send them to the SS storage.

Our solution is not trivial and brings a major challenge: any in-enclave application is limited to 128 MB [31]. If an enclave has a larger size than the EPC, it encrypts unused pages and evicts them to the unprotected memory, suffering a performance penalty.

To prevent frequency analysis attacks, it is necessary to store a table (cache) in the enclave with the frequency of each chunk, that is, the number of times this chunk has already been written. Again,

due to the memory limits of the enclave (128MB) [31], it is not possible to store the frequency of all chunks within the enclave cache, for example, using SHA-256 it is possible to obtain $2^{256}$ hashes. If we consider that there are 100MB available for the enclave cache, only approximately 2M entries can be stored, since each entry takes up 42 bytes (including the hash, a frequency counter, and a reference to the entry in the frequency table). In real storage systems, these 2M entries can represent 5% of the number of existing chunks.

This type of cache was already present in S2DEDUP [30], but, in that system, when the enclave memory limit is reached, the table is reset, preventing the use of deduplication for the chunks from which information was lost. To overcome this problem, our system maintains the frequency of all chunks even when the memory limit of the enclave is reached. The server contains a second encrypted frequency table, outside the enclave, whose contents can only be deciphered by the enclave. The enclave will consult this table (as seen in Figure 4.2) when the memory limit is reached and it is not possible to store the frequency of new chunks. A problem that can arise when using a table in an untrusted zone is that the server can observe the access of the enclave to the entries of this table, and thus also infer the contents through the frequency of accesses [12].

To mitigate this problem, our solution includes a new component within the enclave, called *Access Manager*. This component implements access dispersion policies and cache eviction policies to mask the frequency of accesses to each entry.

## 4.3 Fault Tolerance

Since our system was designed for cloud storage, failures in data centers may affect the functioning of our system. In our design, some information is directly dependent on the enclave and can be lost if the enclave fails. In such a situation, clients would continue to be able to read their data, as reads do not depend on interactions with the the enclave. However, our frequency cache and the secret key used to encrypt the external table would become unavailable. A possible solution is to synchronize multiple enclaves to maintain a copy of this secret key and also replicate the frequency table. The synchronization of this table may not be trivial, since the frequencies may diverge during some time period resulting in different choices for when to perform deduplication inside the enclave.

## 4.4 Privacy Guarantees

To quantify the level of privacy that our *Access Manager* component is able to offer to our table in the untrusted zone, we decided to follow a variant of the privacy preservation criterion used in PraDa [32], called $\alpha$-*privacy*. This criterion ensures that there are always at least $\frac{1}{\alpha}$ chunks with the same frequency.

In our work, we consider a generalization of this criterion, called $(\alpha, \delta)$-*privacy*, which ensures that, given a chunk accessed with a frequency *f*, there are always at least $\frac{1}{\alpha}$ chunks that are accessed with a frequency in the interval $[f - \delta, f + \delta]$. Note that when $\delta = 0$ the $(\alpha, \delta)$-*privacy* is equivalent to $\alpha$-*privacy*.

Figure 4.4 shows an example of the frequency distribution of a data set; this data set contains 5 chunks, some of the chunks share the same frequencies, while others do not.

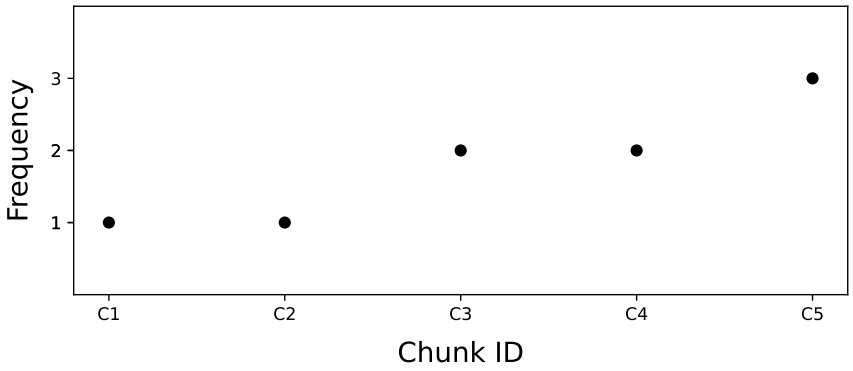


**Figure 4.4:** Example dataset frequency distribution.

When applying our criterion with $\delta = 0$, the value is $\frac{1}{\alpha}$ equals 1, since for each chunk, there is at most 1 chunk with a difference of 0 in frequency. When $\delta = 1$, $\frac{1}{\alpha}$ is equal to 2, since there are at most 2 chunks with a max difference of 1 in frequency. And with $\delta = 2$, $\frac{1}{\alpha}$ is equal to 5, since all chunks have a maximum difference of 2 in frequency.

This generalization allows for capturing scenarios where the frequency of a given chunk cannot be accurately estimated.

## 4.5 Implementation

Here we present the implementation of FH-Dedup. We start by discussing the technologies used to develop our prototype, then we move to the implementation details of the write and read operations, and finally, we discuss the implementation of the *Access Manager* component.

### 4.5.1 Technologies

Our prototype is based on the S2DEDUP code base. This simplified the development and let us focus on the most important aspects of our solution. The prototype is implemented in C and takes advantage of both Intel SGX and the user space Software Performance Development Kit (SPDK) [33, 34], a framework that provides a set of tools and libraries for writing high-performance, scalable, user-mode storage applications.

Our deduplication engine is also implemented as an SPDK virtual block device. It intercepts incoming block I/O requests, performs secure deduplication with user-specified fixed chunk size, and then forwards the request to an NVMe block device or another virtual processing layer depending on the target SPDK deployment. Using NVMe drivers in SPDK is valuable since it provides zero-copy, highly parallel access directly to an SSD from a user-space application.

These requests eventually reach the NVMe driver and storage device, unless intermediate processing removes such a need (for example, repeated writes). In addition, SPDK provides a set of storage protocols that can be stacked on top of the block device abstraction layer. Among them, our work is also implemented using ISCSI targets, allowing clients to remotely access storage servers.

### 4.5.2 Write Operations

When a client wishes to write data to the remote storage server, it begins the process by encrypting each chunk with a key established between the client and the enclave, using a symmetric encryption scheme, in this case, the standard AES scheme with 256-bit keys in XTS mode. XTS was designed as a more robust alternative to other available block cipher modes such as CBC. We choose it because it is a length-preserving scheme (i.e., the ciphertext has the same length as the plaintext), and does not apply chaining, thus supporting random access to encrypted data [35].

We encrypt the data to protect it against attackers eavesdropping on the communication channels. Next, the encrypted data is sent over the network through the ISCSI protocol. The server sends the data to the enclave, which will first decipher the data with the shared key with the client. It will then generate the chunk encryption key using TED's probabilistic scheme, using both the enclave secret and the chunk frequency. The frequency of a chunk can be obtained from the enclave cache or by using the *Access Manager* component, with the policies that we describe below. After generating the key and encrypting the data, the enclave writes the data directly to storage. The enclave also encrypts the chunk encryption key with the shared client key; in order to decrypt the chunk later, this encrypted key is stored inside the server also.

### 4.5.3 Read Operations

When the client retrieves data from the remote server, the server will fetch the encrypted chunks and send them back to the client. For each encrypted chunk, the client will contact the server to retrieve the key (in encrypted form) to decipher the chunk. The client decrypts the key with its shared client key and with the resulting key decrypts the data. This way the enclave is not present in the read operations and no re-encryption of data is done as in S2DEDUP. The read performance comparison between our solution and S2DEDUP is also present in Chapter 5.

### 4.5.4 Access Dispersion Policies

As mentioned previously, our strategy to prevent frequency analysis attacks lies in the *Access Manager* mediating the interaction between the frequency cache residing in the enclave and the encrypted frequency table stored outside the enclave. Its operation is based on the combination of an access dispersion mechanism that is applied whenever the enclave accesses the encrypted frequency table that is in the untrusted zone, and a cache eviction mechanism, which is used to replace entries in the cache when it reaches the maximum size. Next, we describe some of the policies that can be implemented by these mechanisms.

The information exchange mechanism between the enclave cache and the encrypted table on the server can be performed by following several approaches. We consider three access dispersion policies:

- *Direct Access*: The *Access Manager* directly accesses the entry present on the table. This approach has good computational and storage performance, as only the desired server entry is accessed;
- *K-Anonimity Access*: The *Access Manager* queries *K* entries (the desired entry and $K - 1$ random), in random order. This allows for changing the frequency of accesses, causing infrequent chunks to be consulted more often;
- *Bucket Access*: It consists of storing server entries in buckets. To retrieve the desired entry from the server, the *Access Manager* indicates its bucket, not its hash. In this way, the server will send all the entries present in that bucket and will not be able to understand which entry was requested. The bucket of a given entry is given by:

$$\textit{Hashcode}(h) \mod \textit{N} \tag{4.1}$$

  where *N* is the number of buckets, and *Hashcode* is a function that returns an integer given an hash *h* (e.g., $h[0] * 31^{n-1} + h[1] * 31^{n-2} + ... + h[n-1]$, where *h[0]* is the first character of *h*, and *n* is the number of characters). The result identifies the bucket in which the new entry will be placed. This scheme assumes a uniform distribution of entries by buckets and allows chunks with less frequent accesses to share the same bucket with chunks with frequent accesses, thus increasing the access frequency of the former.

### 4.5.5 Cache Eviction Policies

When it is necessary to store the frequency of a chunk but the enclave cache has already reached its maximum capacity, it is necessary to evict one of the entries, to store the new one inside the enclave. For this purpose, we consider 4 eviction policies:

- *Random*: Evict a random element present in the enclave cache;
- *Least Recently Used* (LRU): Evict the element that was accessed the longest in the enclave cache;
- *Less Frequent*: Evict the element with the lowest frequency present in the enclave cache;
- *Least Accessed Externally*: Evict the element with the least access to the encrypted table on the server that is present in the enclave cache.

### 4.5.6 Policies Combination

The different access dispersion policies can be combined with the different cache eviction policies, allowing 12 different modes of operation of the *Access Manager* component. The 12 possible combinations were evaluated, and the results obtained are presented in Chapter 5.

## Summary

In this chapter, we presented the architecture and implementation of FH-Dedup, a deduplication mechanism that leverages trusted execution environments to perform deduplication while storing encrypted files and avoiding frequency analysis attacks. In the next chapter, we present the experimental evaluation of FH-Dedup.

# 5

# Evaluation

## Contents

In this chapter, we evaluate the storage savings provided by our solution, we analyze the privacy guarantees in the access to the encrypted table on the server, and evaluate the performance of read and write operations.

## 5.1 Storage Saving and Privacy Guarantees

In order to evaluate the frequency distribution of access to the server's encrypted table, we created a dataset susceptible to frequency analysis, i.e., a dataset that contains a number of chunks with a high frequency. This dataset was obtained using a data generator that follows a Zipfian distribution (see Figure 5.1) with 10K distinct chunks and 100M accesses. The cache present in the enclave can store up to 500 entries, meaning it can only store 5% of the frequencies of the chunks in the enclave (green area).

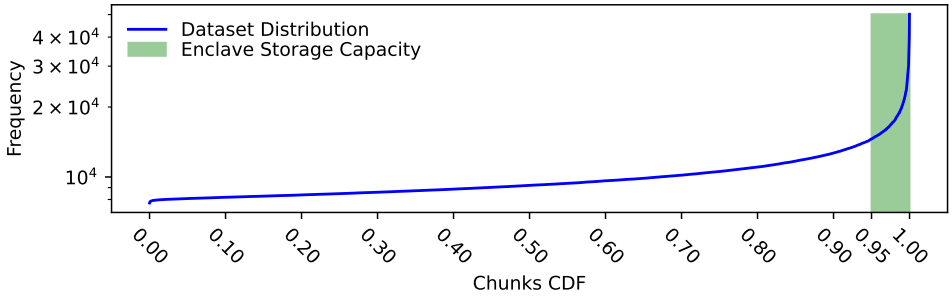


**Figure 5.1:** Frequency distribution of the chunks in the data set.

### 5.1.1 Storage Savings Analysis

The purpose of having an external frequency table is to store the frequencies of all chunks, maximizing the deduplication gain. To verify the deduplication gain, we simulated the storage cost when the S2DEDUP solution is used (with $t$ = 350, when the frequency of a chunk reaches $t$ a new key is used to encrypt it [3]), and our solution (with $t$ = 350). The results are shown in Table 5.1. The analyzed dataset has approximately 381.5 GB (4KB size chunks). These simulations were done by varying the cache size, meaning, the enclave storage capacity.

**Table 5.1:** Storage savings in S2Dedup and in our solution.

| Percentage of entries that can be stored in the enclave | | 5% | 50% | 95% | 100% |
|---|---|---|---|---|---|
| Storage Savings | S2Dedup | 2.7% | 29.3% | 70.1% | 99.74% |
| | FH-Dedup | 99.74% | 99.74% | 99.74% | 99.74% |

As S2DEDUP needs to restart the frequency table within the enclave whenever it reaches its maxi-

mum size, S2DEDUP is limited by the size of the enclave table to offer storage savings through deduplication. On the other hand, FH-Dedup takes advantage of a table in the untrusted zone, overcoming this limitation, thus being able to take full advantage of chunk deduplication. As can be seen in Table 5.1 our solution always offers maximum storage savings, while S2DEDUP has a hard time applying deduplication the smaller the enclave memory. It can only save 2.7% of storage when the enclave can keep only 5% of the entries.

### 5.1.2 Accesses to Untrusted Storage Analysis

To evaluate the information that a malicious server can obtain by observing the accesses to its encrypted table, the execution of each of the combined policies for the dataset presented above was simulated. In addition to the type of access policy used to query the server table, we wanted to understand whether cache eviction policies somehow influence the number of accesses. Figure 5.2 shows the results obtained.

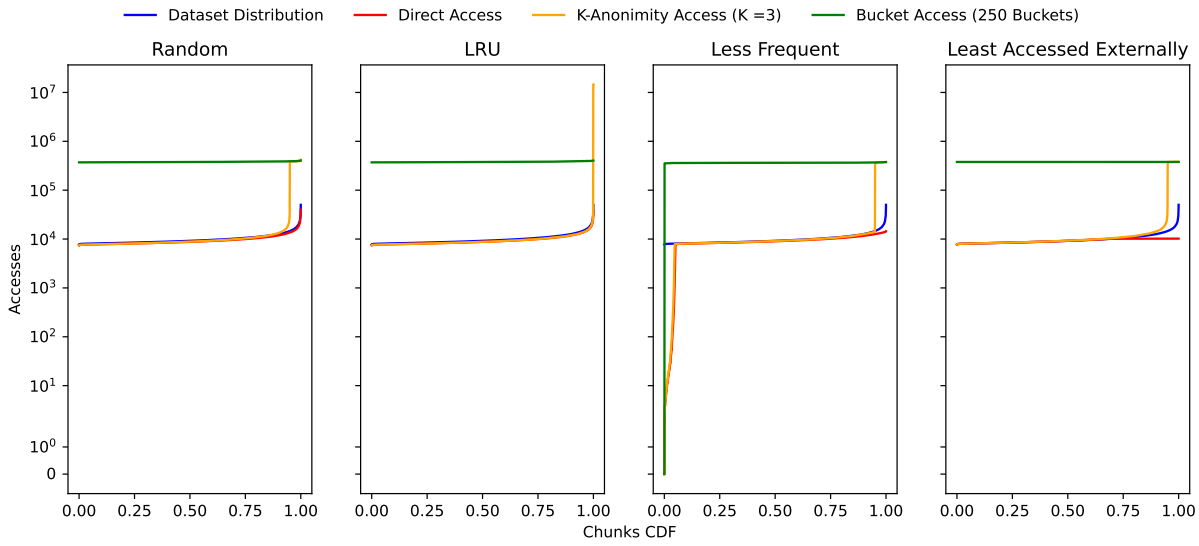


**Figure 5.2:** Distribution of accesses to the server's encrypted table with different cache policies for each approach.

The frequency distribution for the dataset analyzed is shown in blue (the same as shown in Figure 5.1), to represent the effect of policies on access frequency. The distribution of accesses to the encrypted table on the server, using the strategies *Direct Access*, *K-Anonimity Access* and *Bucket Access*, is represented in red, orange, and green, respectively.

When the *Random* cache eviction policy is used, the strategies *Direct Access* and *K-Anonimity Access* show the same type of accesses as the original distribution, however, the accesses are more accented using *K-Anonimity Access*, as the number of hits becomes greater than the original. The access policy *Bucket Access* manages to change the frequency of accesses of chunks but increases the

number of accesses.

For strategies *Direct Access* and *K-Anonimity Access*, the LRU policy does not properly manage the elements that are evicted, as it only takes into account the order in which the chunks are accessed. The access policy *Bucket Access* maintains the previous behaviour.

The eviction policy *Less Frequent* ensures that the most frequent elements are kept within the enclave; however, the less frequent elements will always be evicted, so when they are needed, it will be necessary to retrieve their counter from the frequency table outside the enclave. With access policies *Direct Access* and *K-Anonimity Access*, this reduces the number of accesses to some elements. The access policy *Bucket Access* shows a reduction in the number of accesses, but this number continues to be much higher than the other approaches.

Finally, the eviction policy *Least Accessed Externally* is the one that best normalizes the frequency of accesses to each element in the server table. This can be observed especially with strategies *Direct Access* and *Bucket Access*, which are closer to a straight line. This effect is not easy to achieve with the *K-Anonimity Access* approach because, despite dumping the least accessed elements in the untrusted table, their accesses are random, causing some elements to continue to be accessed many more times.

### 5.1.3 Privacy Guarantees Analysis

In order to evaluate the $(\alpha, \delta)$-*privacy* offered by each combination, the criterion presented in Section 4.4 was applied to the dataset for different values of $\delta$. The results are shown in Figure 5.3.

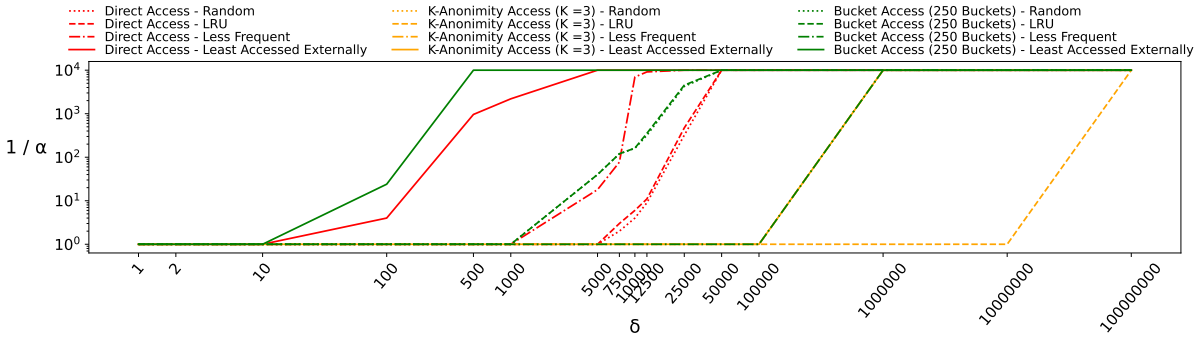


**Figure 5.3:** Privacy guarantees offered by each combination.

The calculation of $\alpha$ was performed using the function *f(X,δ)*, which calculates $\alpha$ for a given $\delta$ in a dataset *X*. Note that the value of $\alpha$ falls between $1/N$ and 1, where *N* is the number of distinct chunks. The higher the value of $1/\alpha$ and the lower the value of $\delta$, the greater the privacy offered by the combination.

The combination of the *Bucket Access* access policy with the *Least Accessed Externally* eviction policy has the best $(\alpha, \delta)$-*privacy*. With $\delta$ equal to 100, we have $\alpha = 1/24$. Given that $\frac{1}{\alpha} = 24$, this

means that for each chunk we have 24 other chunks with a maximum difference of 100 in frequency. With $\delta$ equal to 500, all 10K chunks have a maximum frequency difference of 500. The second best combination is *Direct Access* with the *Least Accessed Externally* cache eviction policy: with $\delta$ equal to 100, $\frac{1}{\alpha}$ equals 4; with $\delta$ equal to 500, $\frac{1}{\alpha}$ equals 960; with $\delta$ equal to 1K, $\frac{1}{\alpha}$ equals 2197; and with $\delta$ equal to 5K, $\frac{1}{\alpha}$ is 10K.

The other combinations are able to achieve $\frac{1}{\alpha}$ equal to 10K but for much larger values of $\delta$, making it easier to distinguish the chunks between them, making the use of these combinations less secure.

## 5.2 Performance Evaluation

To evaluate the performance of FH-Dedup, we carried out experiments to understand the impact of the use of the *Access Manager* on the performance of write operations and the intervention of the enclave in read operations. Our testbed consists of an Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz with 16GB of RAM and SGX.

### 5.2.1 Write Operations

Our solution allows us to maintain deduplication gains while providing privacy guarantees: however, to maintain this effect, the *Access Manager* has to interact with the encrypted frequency table of the server to store information about chunks that it cannot store in the enclave cache.

To evaluate the impact of the encrypted table in an untrusted environment, we performed multiple write operations with 4KB chunks, more specifically we only compare the cost of enforcing each policy. By this, we mean that we measure operations that include hash calculations, OCALLs, and table accesses since this is where our policies and S2Dedup differ. For these write operations, we exclude operations that require chunk encryption or memory copies, since such operations are equal in any policy. We made this choice to better understand the performance overhead introduced by each combination of policies.

We compared each of our access combinations (except LRU, not implemented) with S2DEDUP; the results are shown in Figure 5.4.

In this experiment, we use a cache inside the enclave that supports 8196 entries and gradually increases the dataset size, composed only of write operations, which are those that make the enclave interact with the external frequency table. Another important aspect is that we tested the worst case where all writes consist of distinct chunks, which causes FH-Dedup to interact with the server's encrypted frequency table every time the cache is full and a new chunk is written.

While the number of writes is up to 8196 ($2^{13}$ in the figure), our solution behaves exactly like S2DEDUP, i.e., information about chunks is written directly to the enclave cache. When the number

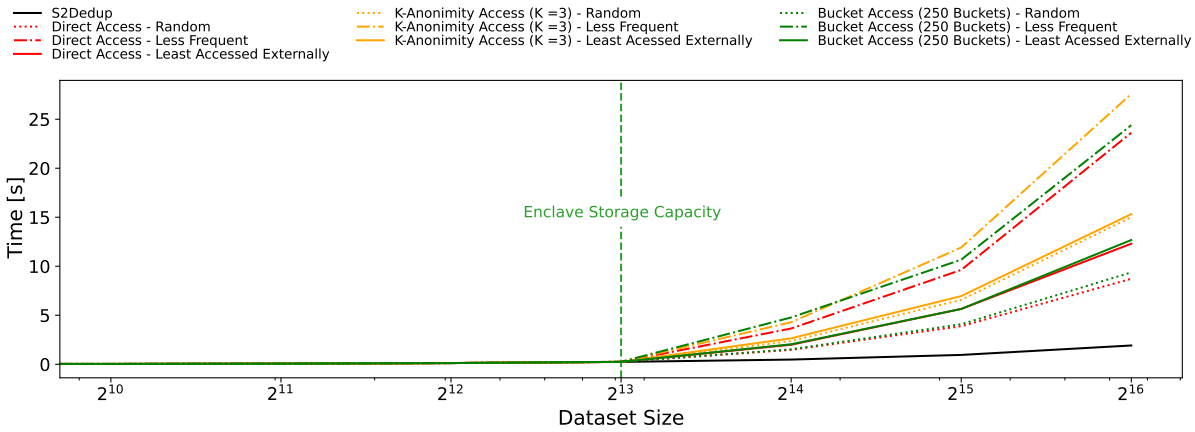


**Figure 5.4:** Cost to enforce each policy for a new chunk.

of writes is greater than 8196, communication with the server is required since the cache no longer allows for the storage of more information.

There are two factors in FH-Dedup that increase overhead. The first is the access policy, which can contribute to a larger number of OCALLs and memory allocation; the second one is the cache eviction policy, which can cause evictions from *Access Manager* to take a longer period of time.

The *Direct Access* policy requires OCALLs to retrieve the desired entry from the server frequency table, and an OCALL to write the new (or updated) information about the evicted entry.

The *K-Anonimity Access* policy requires the same number of OCALLs as the *Direct Access* policy multiplied by *K*, we will consult *K* entries in the server, and evict one entry.

Finally, the *Bucket Access* policy retrieves all the entries in the desired bucket, since we ask for information from the server by sending a bucket identifier and not a hash through an OCALL. It scans all elements until it finds the desired entry and decrypts its counter. Later, it will evict one entry from the enclave cache.

For all the policies, we also need to allocate memory for the information to be returned from the server, and we need to decrypt the desired entries counters.

We can see in the figure that the random cache eviction strategy is the least expensive. This happens because enclave entries do not need to be sorted, we just generate a random value in the range $[0, 8196]$ and select the entry in the generated index.

Other cache eviction strategies are more expensive because they require entries to be sorted by frequency or external hit counts in the server frequency table.

The least external access strategy outperforms the least frequency strategy because the sorting process takes less time. This happens because external accesses to server frequency table entries do not change while they are in the enclave cache, as opposed to an entry frequency, which gets updated for elements inside the cache.

*K-Anonimity Access* is the slowest access strategy due to our implementation. Every time a new entry needs to be added, it needs to pick $K$-1 items from the enclave cache in a random fashion, so it must first create an array with all the elements of the cache, and then select these random elements. Later, when evicting an entry, it must pick another random element to evict, so it must bring all cache elements again into an array and then randomly select one element. The other access strategies only require this last step.

The implementation of the *K-Anonimity Access* policy could be improved using other data structures, however, this improvement could not be implemented in time.

### 5.2.2 Read Operations

To observe the performance difference of read operations in S2DEDUP and FH-Dedup, we performed several read operation experiments, we varied the number of 4KB chunks read and measured the amount of time required to read and decrypt these chunks. We also use the same type of symmetric cipher as S2DEDUP, AES-256-XTS. The results are shown in Figure 5.5:

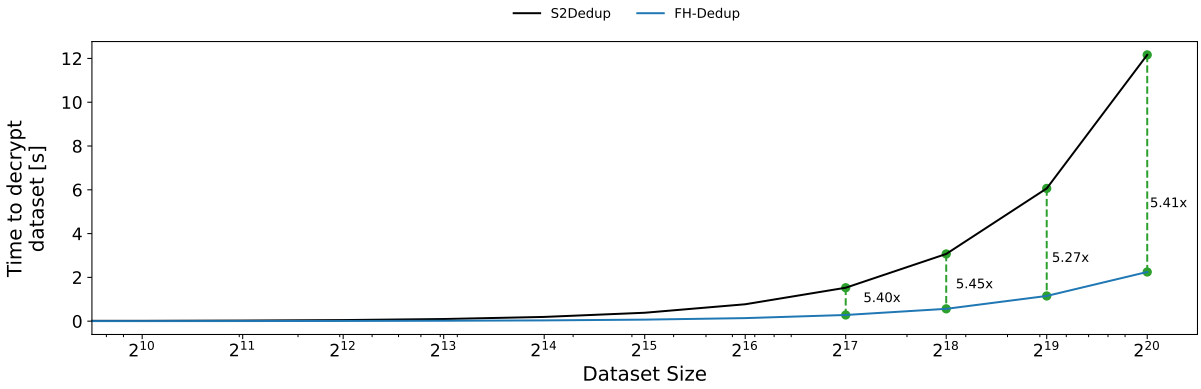


**Figure 5.5:** Time cost to read and decrypt different datasets.

From the results obtained, we conclude that FH-Dedup, on average, performs read operations 5.38 times faster than S2DEDUP.

To cross-check our results are trustworthy, we tested the speed of this cipher suite using OpenSSL's *speed* command. It takes 3 seconds to encrypt/decrypt 3,147,723 4KB chunks. This means that it can encrypt/decrypt an average of 1,049,241 4KB chunks per second. FH-Dedup takes approximately 2.25 seconds to read and decrypt 1,048,576 4KB chunks, making the result credible.

In S2DEDUP, every time a client requests an encrypted chunk, this encrypted chunk must be deciphered in the enclave using the enclave key, and ciphered with the client key, later, the client will decipher the encrypted chunk with its key, finally retrieving the original plaintext chunk. This means that every time the client requires the same chunk, it will be necessary to perform two decryption operations and one

encryption operation on chunks.

In FH-Dedup there is no need to communicate with the enclave; the client retrieves the encrypted key recipe from the server and deciphers it with its key, and from the key recipe, the client can retrieve the key needed to decipher the chunk. Due to this approach, there are only two decryption operations, one necessary to obtain the key for decryption and one to decrypt the chunk.

This makes FH-Dedup reads more performant and scalable than S2DEDUP, removing bottlenecks in the enclave during reads. Typically, reads have a 70/30 ratio compared to writes, making them the most common operations in memory systems [36].

## 5.3 Discussion

It is observed that the access policy *Bucket Access* is the one that manages to keep the access frequencies of the chunks close to each other, regardless of the cache eviction policy used, making it more difficult to distinguish the chunks. This happens because less frequent chunks can be placed in the same bucket as more frequent chunks, changing their access frequency, and making it more difficult to perform frequency analysis attacks. However, this access policy is computationally more expensive than the *Direct Access* policy, and also increases the number of accesses by more than an order of magnitude. With the *Least Accessed Externally* policy, the *Direct Access* access policy is also able to keep the access frequencies of the chunks close to each other, but with a lower number of accesses. However, using this eviction policy implies higher memory usage within the enclave, which leads to a smaller number of entries that can be stored in the enclave. We can also observe that FH-Dedup creates overhead when it must retrieve information from the encrypted table during write operations; nevertheless, in the most common operations, the read operations, we achieve faster reads than systems that require the enclave interaction, specifically, we, are on average 5.38 times faster than S2DEDUP.

## Summary

In this chapter, we present our evaluation, starting with the storage savings provided by our solution. We can observe that our system always guarantees the same deduplication gains, despite the size of the enclave cache. We then discuss the number of accesses given by each of the policy combinations, and finally, we evaluated the security provided by each of the combinations.

We also evaluated the performance of read and write operations, we see that our system creates overhead when it must retrieve information from the encrypted table, nonetheless, the most common operations, the read operations, are on average 5.38 times faster than S2DEDUP.

In the next chapter, we present our conclusions as well as current system limitations and future work for overcoming these limitations.

# 6

# Conclusion

## Contents

In this chapter, we present our conclusion and our view on the system's limitations and what can be done in the future to address these limitations.

## 6.1 Conclusions

With the rapid growth of data, cloud storage systems have become popular. To allow the storage of duplicate data without compromising storage efficiency, new techniques were introduced.

Data deduplication is a technique that reduces the amount of redundant data held in a cloud storage service. If the data is already stored, a pointer to that copy of the data is created, rather than storing additional copies of the data.

To address users' privacy, encrypted deduplication was created, combining deduplication with privacy, allowing users to store data encrypted while allowing storage systems to apply deduplication.

Several attacks can occur in these types of systems, one of them, being frequency analysis, which aims to find out which data is being stored encrypted by users.

In this work we presented an Intel SGX-based solution that is capable of supporting deduplication while maintaining strong levels of privacy, preventing frequency analysis attacks. In this context, we propose and compare different combinations of cache strategies and policies to prevent frequency analysis attacks.

Our evaluation demonstrates that the combination of different caching strategies and policies influences the privacy guarantees and the number of accesses to the server (encrypted) frequency table. Our performance evaluation also let us understand the overhead added by our solution in write operations when new elements cannot be stored inside the enclave cache, nonetheless, our read operations are faster than the state-of-the-art systems, in particular, on average 5.38 times faster than S2DEDUP.

Furthermore, the storage savings in our system are constant, regardless of the size of the cache present in the enclave, having greater storage savings than the most recent state-of-the-art solution, which also uses chunk frequency to apply encrypted deduplication inside an enclave. This way, we are enabling storage efficiency, with a commitment to privacy.

## 6.2 System Limitations and Future Work

Our work shows that it is possible to use TEEs as secure mediators in encrypted deduplication systems while maintaining deduplication gains even with memory limitations. Our solution uses an external encrypted frequency table to keep track of chunk frequencies that can no longer be stored inside the enclave cache. However, our solution with the use of OCALLs and different access and cache eviction policies has a performance penalty during write operations, we consider that better access strategies

and cache eviction policies should be discussed and implemented.

Also, our current architecture allows for one server and multiple clients. In a more realistic environment, there should be multiple servers that a client can contact to store/retrieve data. For this reason, we consider that an improvement is having multiple servers that communicate with each other the frequency of their chunks through a lazy replication communication protocol such as gossip. Although this design seems simple it is not trivial; remember the chunk frequency in the server is encrypted, so the chunk frequencies must first be decrypted in the enclaves and later be updated.

# Bibliography


[1] Dropbox, Available at https://dropbox.com/, Accessed: 2022-10-31.

[2] Google Drive, Available at https://www.google.com/drive/, Accessed: 2022-10-31.

[3] J. Li, Z. Yang, Y. Ren, P. Lee, and X. Zhang, "Balancing storage efficiency and data confidentiality with tunable encrypted deduplication," in *European Conference on Computer Systems*, Heraklion, Greece, Apr. 2020.

[4] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Computing Surveys*, vol. 47, pp. 1–30, 2014.

[5] D. Meyer and W. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage*, vol. 7, pp. 1–20, 2012.

[6] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *International conference on File and Storage Technologies*, San Jose (CA), USA, Feb. 2012.

[7] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, "Single instance storage in windows 2000," in *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle (WA), USA, Aug. 2000.

[8] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms," Hewlett-Packard Labs, Tech. Rep. 30, 2005.

[9] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams," in *8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose (CA), USA, Feb. 2010.

[10] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Transactions on Storage*, vol. 2, pp. 424–448, 2006.

[11] G. Lu, Y. Jin, and D. H. Du, "Frequency based chunking for data de-duplication," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Miami Beach (FL), USA, Aug. 2010.

[12] M. Jurado and G. Smith, “Quantifying information leakage of deterministic encryption,” in *International Conference on Cloud Computing Security Workshop*, London, United Kingdom, Nov. 2019.

[13] M. Bellare, S. Keelveedhi, and T. Ristenpart, “Message-locked encryption and secure deduplication,” in *International Conference on the Theory and Applications of Cryptographic Techniques*, Athens, Greece, May 2013.

[14] D. Harnik, B. Pinkas, and A. Shulman-Peleg, “Side channels in cloud services: Deduplication in cloud storage,” *IEEE Security & Privacy*, vol. 8, pp. 40–47, 2010.

[15] S. Keelveedhi, M. Bellare, and T. Ristenpart, “Dupless: Server-aided encryption for deduplicated storage,” in *Security Symposium USENIX Security*, Washington (D.C.), USA, Aug. 2013.

[16] GeeksforGeeks, “Caesar cipher in cryptography,” Available at https://www.geeksforgeeks.org/caesar-cipher-in-cryptography, Accessed: 2022-10-31.

[17] J. Li, P. P. Lee, C. Tan, C. Qin, and X. Zhang, “Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses,” *ACM Transactions on Storage*, vol. 16, pp. 1–30, 2020.

[18] Z. Ning, J. Liao, F. Zhang, and W. Shi, “Preliminary study of trusted execution environments on heterogeneous edge platforms,” in *Proceedings of the 1st ACM/IEEE Workshop on Security and Privacy in Edge Computing*, Bellevue (WA), USA, Oct. 2018.

[19] J.-E. Ekberg, K. Kostiainen, and N. Asokan, “The untapped potential of trusted execution environments on mobile devices,” *IEEE Security & Privacy*, vol. 12, pp. 29–37, 2014.

[20] SGX101, “Overview - SGX 101,” Available at https://sgx101.gitbook.io/sgx101/sgx-bootstrap/overview, Accessed: 2022-10-31.

[21] C. Correia, M. Correia, and L. Rodrigues, “Omega: a secure event ordering service for the edge,” in *IEEE/IFIP International Conference on Dependable Systems and Networks*, Valencia, Spain, Jun. 2020.

[22] M. Naor and O. Reingold, “Number-theoretic constructions of efficient pseudo-random functions,” *Journal of the ACM (JACM)*, vol. 51, pp. 231–262, 2004.

[23] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko, “The one-more-rsa-inversion problems and the security of chaum’s blind signature scheme.” *Journal of Cryptology*, vol. 16, p. 185–215, 2003.

[24] J. Camenisch, G. Neven, and A. Shelat, “Simulatable adaptive oblivious transfer,” in *International Conference on the Theory and Applications of Cryptographic Techniques*, Barcelona, Spain, May 2007.

[25] D. Chaum, “Blind signatures for untraceable payments,” in *CRYPTO*, Santa Barbara, CA, USA, Jan. 1983.

[26] G. Cormode and S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications,” *Journal of Algorithms*, vol. 55, pp. 58–75, 2005.

[27] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with haven,” *ACM Transactions on Computer Systems*, vol. 33, pp. 1–26, 2015.

[28] Y. Ren, J. Li, Z. Yang, P. P. Lee, and X. Zhang, “Accelerating encrypted deduplication via SGX,” in *USENIX Annual Technical Conference*, Remotely, Jul. 2021.

[29] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, “Proofs of ownership in remote storage systems,” in *Proceedings of the 18th ACM conference on Computer and communications security*, Chicago (IL), USA, Oct. 2011.

[30] M. Miranda, T. Esteves, B. Portela, and J. a. Paulo, “S2Dedup: SGX-enabled secure deduplication,” in *International Conference on Systems and Storage*, Haifa, Israel, Jun. 2021.

[31] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig, “Benchmarking the second generation of intel SGX hardware,” in *International Conference on Management of Data*, Philadelphia (PA), USA, Jun. 2022.

[32] B. Dong, R. Liu, and W. H. Wang, “PraDa: Privacy-preserving data-deduplication-as-a-service,” in *International Conference on Conference on Information and Knowledge Management*, Shanghai, China, Nov. 2014.

[33] SPDK, “Storage performance development kit,” Available at https://spdk.io/, Accessed: 2022-10-31.

[34] SPDK, “Spdk github,” Available at https://github.com/spdk/spdk, Accessed: 2022-10-31.

[35] M. Dworkin, “SP 800-38E. recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices,” Available at https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=904691, Accessed: 2022-10-31.

[36] Live Optics, “Read / write ratio,” Available at https://support.liveoptics.com/hc/en-us/articles/229590547-Live-Optics-Basics-Read-Write-Ratio, Accessed: 2022-10-31.